Discovering loners and phantoms in commit and issue data

Schermann, Gerald ; Brandtner, Martin ; Panichella, Sebastiano ; Leitner, Philipp ; Gall, Harald

Abstract: The interlinking of commit and issue data has become a de-facto standard in software development. Modern issue tracking systems, such as JIRA, automatically interlink commits and issues by the extraction of identifiers (e.g., issue key) from commit messages. However, the conventions for the use of interlinking methodologies vary between software projects. For example, some projects enforce the use of identifiers for every commit while others have less restrictive conventions. In this work, we introduce a model called PaLiMod to enable the analysis of interlinking characteristics in commit and issue data. We surveyed 15 Apache projects to investigate differences and commonalities between linked and non-linked commits and issues. Based on the gathered information, we created a set of heuristics to interlink the residual of non-linked commits and issues. We present the characteristics of Loners and Phantoms in commit and issue data. The results of our evaluation indicate that the proposed PaLiMod model and heuristics enable an automatic interlinking and can indeed reduce the residual of non-linked commits and issues in software projects.

DOI: https://doi.org/10.1109/ICPC.2015.10

Discovering Loners and Phantoms in Commit and Issue Data

Gerald Schermann, Martin Brandtner, Sebastiano Panichella, Philipp Leitner, and Harald Gall
University of Zurich, Department of Informatics, Switzerland
{schermann, brandtner, panichella, leitner, gall}@ifi.uzh.ch

***Abstract*—The interlinking of commit and issue data has become a de-facto standard in software development. Modern issue tracking systems, such as JIRA, automatically interlink commits and issues by the extraction of identifiers (e.g., issue key) from commit messages. However, the conventions for the use of interlinking methodologies vary between software projects. For example, some projects enforce the use of identifiers for every commit while others have less restrictive conventions. In this work, we introduce a model called *PaLiMod* to enable the analysis of interlinking characteristics in commit and issue data. We surveyed 15 Apache projects to investigate differences and commonalities between linked and non-linked commits and issues. Based on the gathered information, we created a set of heuristics to interlink the residual of non-linked commits and issues. We present the characteristics of *Loners* and *Phantoms* in commit and issue data. The results of our evaluation indicate that the proposed *PaLiMod* model and heuristics enable an automatic interlinking and can indeed reduce the residual of non-linked commits and issues in software projects.**

I. Introduction

The interlinking of commit and issue data plays an important role in software development, during the release planning or the bug triaging. It has become a de-facto standard in software projects, which is reflected in guidelines of large open-source communities, such as Apache: "*You need to make sure that the commit message contains [...] and ideally a reference to the Bugzilla or JIRA issue where the patch was submitted.*" [1].

However, sometimes developers violate guidelines and commit changes without issue reference [2], [3]. For this reason, various research studies investigated possible ways to recover missing links (e.g., [2], [4], [5]). Researchers proposed heuristics (e.g., [6], [7]) to automatically recover missing links by mining issue tracking platforms and commit messages in logs of version control systems (VCS). These heuristics rely on keywords, such as "*bug*" or "*fix*", and issue ids, such as "*#41*", in commit messages. However, these approaches are often not sufficient to detect all of the missing links between issues and commits. Recent research (e.g., [8], [9]) defined more complex approaches based on text-similarity, which are able to recover a higher percentage of missing links. The former research has in common that it is build on a scenario where no explicit interlinking of commit and issue data is available. In difference to modern issue tracking systems, such as JIRA, which support an automated interlinking based on issue keys in commit messages. Hence, it seems valuable to take this information into account rather than starting from no links. The availability of an explicit interlinking enables new research directions. For example, the profiling of developers based on their activity in the VCS and the issue tracking platform [10].

In this work, we investigate the characteristics of data interlinking in development environments that use a modern issue tracking platform with interlinking capabilities. For our investigation, we introduce a model called *Partial Linking Model (PaLiMod)* to support the integration of commit and issue data. On top of this model, we surveyed the interlinking of commit and issue data within 15 Apache projects that use JIRA as issue tracking platform. Based on the gathered information, we derived the characteristics of two interlinking scenarios called *Loners* and *Phantoms*. Loners in the context of our work describe single commits that have no link to the addressed issue. In case of a Loner, no other commit addresses the same issue, which is the major difference to Phantoms. Phantoms are commits without link occurring in a series of commits that address a certain issue. For example, a developer commits three changes addressing an issue, but only the last provides an issue key in its commit message. Based on the investigated characteristics, we propose heuristics to automatically detect *Loners* and *Phantoms* for reducing the residual of non-linked commit and issue data.

The main contributions of this paper are as follows:

- *A formal model to investigate the partial interlinking of commit and issue data.*
- *A survey to investigate the practice of commit and issue interlinking in 15 Apache projects.*
- *A set of heuristics to automatically handle missing links in partially linked commit and issue data.*
- *A prototypical implementation of the model and the interlinking heuristics.*

The results of our survey showed that on average 74.3% of the commits contain issue keys, but only 49.6% of the resolved issues are linked to a commit. Most of the linked commits (37.4%) are combined commits, which means they contain source code changes and test code changes. The largest group of the non-linked commits (47.0%) are those commits not addressing any source code change.

We evaluated the proposed heuristics in a series of simulated project scenarios that contain different residuals of non-linked commits and issues. For the simulation, we removed links between commits and issues that were explicitly linked by an issue key in the commit message. This evaluation approach

is enabled by the PaLiMod model and allows for an accurate performance evaluation of the proposed heuristics.

The results of the heuristic evaluation showed that our approach can achieve an overall precision of 96% with a recall of 92% in case of the Loner heuristic, and an overall precision of 73% with a recall of 53% in case of the Phantom heuristic.

The remaining paper is structured as follows. We introduce the approach in Section II, followed by an overview of the proposed PaLiMod model and the results of the survey in Section III. In Section IV, we introduce the interlinking heuristics, and the evaluation in Section V. We discuss results in Section VI. Section VII covers the threats to validity. The most relevant related work is presented in Section VIII. The paper is concluded with our main findings in Section IX.

II. Approach

The aim of this paper is the investigation of commit and issue data interlinking characteristics in software projects that have guidelines (e.g., [1]) to link source code contributions to issues. Based on the findings, we derive a set of heuristics to enable an automatic interlinking of the residual of non-linked commit and issue data within such software projects. For these investigations, we introduce a model to support the integration and analysis of commit and issue data. We investigate the extent to which developers follow such interlinking guidelines and the cases in which a guideline is not followed. The insights which we found are used to come up with a selection of heuristics to address cases in which an interlinking guideline was not followed.

We address the aim of our approach with the following two research questions:

RQ1: *What are the characteristics of interlinked commit and issue data in projects that have interlinking guidelines?*

RQ2: *How can an automatic interlinking of the residual of non-linked commit and issue data be enabled based on such characteristics?*

Figure 1 depicts the approach which we followed to answer the research questions of this work.

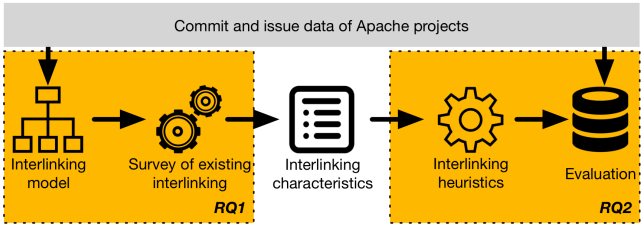

Fig. 1. Overview research approach

In a first step, we defined the Partial Linking Model (PaLiMod), which is used to integrate and store the extracted commit and issue data for the project survey. Based on the integrated data of 15 Apache projects, we extracted a set of interlinking characteristics that occurred in all of the analyzed projects. In a further step, the resulting characteristics of non-linked commits and issues are used to define heuristics to enable an automatic residual interlinking. Finally, we evaluate the performance of the proposed interlinking model and heuristics with a scenario-based approach.

III. Partial Linking Model & Project Survey

The proposed PaLiMod model is used for a project survey on the commit and issue interlinking in 15 Apache projects, which are listed in Table I.

TABLE I
Development activity in Apache projects between September '13 and September '14 with number of commits, number of resolved issues, and the number of issues that are linked to one (1:1) or multiple commits (1:n).

ID	Project	Commits	Issues	Links 1:1	Links 1:n
P1	ActiveMQ	1058	901	165	166
P2	Ambari	5012	4249	2794	885
P3	Camel	3994	1272	161	610
P4	CXF	3792	968	25	137
P5	Felix	983	799	92	33
P6	Hadoop	1367	1998	62	125
P7	HBase	4110	4897	389	829
P8	Hive	2234	3926	1343	281
P9	Jackrabbit Oak	3781	1477	90	72
P10	Karaf	2059	1020	137	385
P11	PDFBox	1327	1100	99	120
P12	Sling	4174	1086	109	256
P13	Spark	5482	2748	452	706
P14	Stanbol	517	305	32	110
P15	Tika	332	347	53	24
	Mean	2681	1806	400	316
	Standard Deviation	1665	1412	716	287

A. PaLiMod - Partial Linking Model

The aim of the proposed PaLiMod model is to capture partly linked commit and issue data in software projects. The model relies on a graph-based schema to describe relationships of *Subjects*. A subject in the context of the PaLiMod model can be a change set, a change request, a person, or a resource. Some of the subjects have sub-subjects, for example, bug or feature to allow for a more precise classification of the according dataset. The categorizations are described in detail in Section III-A1 to III-A3. The subjects represent nodes in the proposed graph. The relationships of each subject are described by so-called *Annotations*. The proposed model offers annotations for relationships between commit, issue, person, and resource subjects. Figure 2 depicts an overview of the RDF based model with its subjects and annotations.

A major difference of the proposed PaLiMod model compared to existing approaches is the support of mixed environments that contain commit and issue artifacts that are partly explicit linked instead of environments without any linking. Additionally, the graph-based manner of the PaLiMod model enables the definition of multiple annotations between two subjects in concurrent.

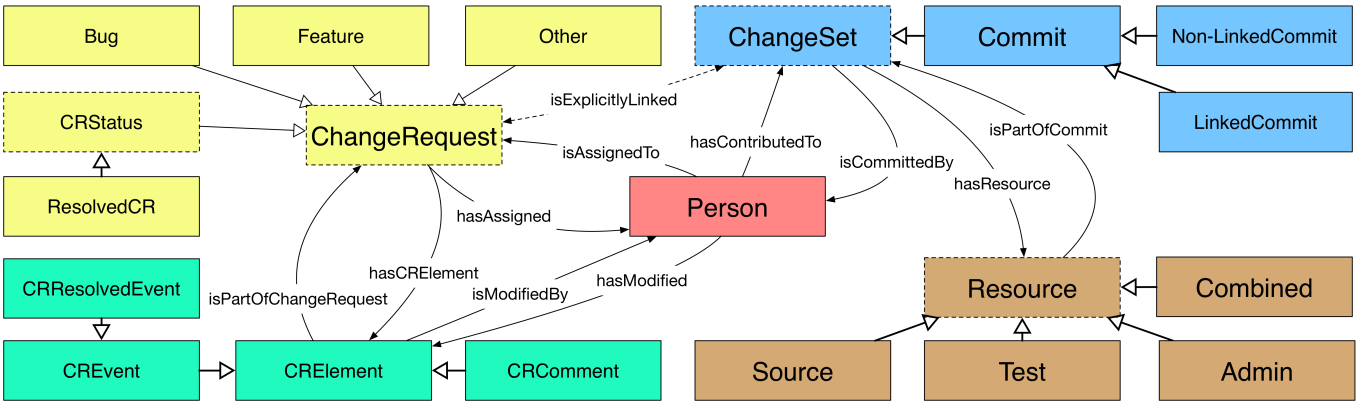

Fig. 2. PaLiMod model

The listed subjects and annotations represent an initial set of possible nodes and edges, which can be even expanded for other purposes as well. For example, by incorporating additional change request states in *CRStatus*, such as closed and reopened. Representing the state as an own subject instead of a change request's property provides advanced querying options (multiple inheritance). A prototypical implementation of the PaLiMod model based on OWL/RDF is available for download on the project's website [11].

1) Classification of Commits: For the classification of commits, we propose the categories *linked* commits and *non-linked* commits. Linked commits are commits containing a reference (e.g., issue key) to an issue in the issue tracking system. Commits without such a mapping are called non-linked commits. Basically, a commit can tackle multiple issues at once, but state-of-the-art issue tracking platforms, such as JIRA, support a single issue key per commit message only. Thus, a single commit is either linked to a certain issue or not. On the other hand, a single issue can have multiple linked commits.

2) Classification of Resources: For the classification of resources, we propose the categories *source*, *test*, *admin* commits, and a *combination* of them (see Table II). This classification is derived from the standard directory layout used in Maven projects.[1] Thus, this survey focuses on Java projects using Maven and following Maven's standard directory layout.

TABLE II
RESOURCE CLASSIFICATION

Classification	Description
Source	All associated resources are source code files.
Test	All associated resources are test code files.
Admin	All files are neither source, nor test code files.
Combination	The associated resources are a mixture of source code, test code, and other files.

[1]http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

A commit is classified as a source commit if all of its created, modified or deleted resources are source code files. A file is considered a source code file if its path in the project's directory structure contains the snippet *src/main/java*, which corresponds to Maven's standard directory layout. Similarly, the snippet *src/test/java* is used to identify test code files. Therefore, a commit is called a test commit if all of its addressed files are test code files. All other files, which are neither source, nor test code files are called admin files (e.g., changes in the pom.xml). Hence, a commit with only such admin files is referred to as an admin commit. Finally, commits which contain a mixture of source, test and admin files are summarized in the combination category.

3) Classification of Issues: We categorize issues based on the issue type information extracted from the issue tracker. As different naming schemes are used among the analyzed projects in respect to issue types, we assigned them to the simplified categories *Bug*, *Feature*, and *Other*. For example, issues with types *Bug* and *Epic* were assigned to the *Bug* category, in addition *Enhancement* and *Wish* are considered for the *Feature* category. Table III provides the complete mapping of issue types to the categories used throughout this work.

TABLE III
ISSUE CLASSIFICATION

Classification	Description
Bug	All issues with associated issue type *Bug* or *Epic*.
Feature	All issues with associated issue type *Enhancement*, *Wish*, *Task*, *Feature Request*, *Subtask*, *Improvement*, or *Feature*.
Other	All remaining issues.

B. Apache Project Survey

In our project survey, we use data from VCSs (Git) and issue tracking platforms (JIRA) of 15 Java-based Apache projects (see Table I). All projects are characterized by a high activity in the VCS and the issue tracking platform within the last year. We first selected projects with high commit activity provided

by OpenHUB.[2] In a second step, we looked at the projects' issue trackers and reduced our selection to 15 projects with high activity in both domains. A project's activity is considered high if there are at least 20 commits and 20 issues per month. We extracted commit and issue data of the period between September '13 to September '14 and stored it into instances of the PaLiMod model. The full datasets are available for download on the project's website [11].

1) Survey Results: Table IV shows the distribution of linked and non-linked commits of the analyzed projects. In addition, the shares of (sub-) categories source, test, admin, and combination are provided below the overall linked and non-linked shares. The ratio section in the table provides the ratio of non-linked to linked, including the sub-categories as well. For example, the portion of linked source commits is twice as high (ratio 0.5) as the portion of non-linked source commits in the Apache ActiveMQ project (P1). In the same fashion as Table IV, Table V shows a distribution between linked and non-linked as well, but from the issues perspective. Again the shares of (sub-) categories are listed below the overall linked and non-linked results alongside with the ratio information.

The survey of Apache projects showed that interlinking of commit and issue data is used by all of the analyzed projects. In the majority of the analyzed projects (12 of 15), the number of commits linked to an issue is higher than the number of commits without link. However, this result is not reflected in the number of linked issues, as the majority of issues is not linked to a commit in the analyzed projects (8 of 15). One reason for such a deviation is the existence of issue entries (e.g., tasks) that do not require a change in the VCS. However, an indicator against this assumption is the rather high amount of non-linked *Bug* issue entries. Another reason for the missing one-to-one mapping can be the circumstance that an issue fix requires more than one commit and thus, developers miss to report all the changes related to this issue. For example, the bug with issue key HIVE-6782[3] of the project Hive (P8) required multiple commits until it was resolved.

A closer look on the non-linked commits shows that the mean number of *test* and *admin* commits together build their largest share (57.3%). The ratio values of the commit data reflect that circumstance as well, because the highest ratios between the linked and non-linked values are in the categories *test* and *admin*. This can be an indicator that either no issue entries exist or that committers do not take care of referencing such kind of changes to an issue.

The survey data of the issues shows almost no difference in the distribution in the categories of linked and non-linked data. This is also reflected in the ratio values that indicated that the biggest deviation can be found in the *other* category. It is very likely that the reason for this variation can be found in the used issue workflow within a software project. For example, some projects introduce issue types in addition to the default set of types provided by the issue tracker.

2) Combined Commits: The commit overview in Table IV shows that in six projects the highest number of linked commits is caused by so-called combined commits. In case of linked commits, most of the combined commits contain source code and test code changes (47.4%). Another share of combined commits (27.8%) consists out of a composition of source code, test code, and configuration changes. Followed by source code and configuration changes (18.4%), and test code and configuration changes (6.4%).

The combined commits of non-linked issues have a different constellation compared to the linked ones. The majority (52.2%) of the combined commits in this case contain source code, test code, and configuration changes. Commits consisting of source code and test code changes (26.2%) provide the second largest share, followed by source code and configuration changes (11.5%), and test code and configuration changes (10.1%).

3) Interlinking Characteristics: During the survey, we also investigated interlinking characteristics in the analyzed Apache projects. We found two characteristics that occur in all of the projects. The first interlinking characteristic that we found reflects the situation of one commit that is associated to one issue and vice-versa. We call it *Loner*, because in case of a missing interlinking the according commit or issue is without any relationship. An example is the issue AMBARI-3412[4] of Ambari (P2) and the associated, but not linked commit f2146a41[5]. The problem is that the commit message contains the issue key in the wrong format and therefore the link is not detected. We call the second interlinking characteristic *Phantom*. In case of this characteristic, an issue has multiple associated linked commits, but there are further commits that are associated with, but not linked to the issue. These associated, but not linked commits are called Phantoms, because the link is not explicitly established and therefore not visible. An example for a Phantom is CAMEL-7354[6] and the not linked commit 14cd8d6b[7]. These characteristics represent an initial set of interlinking characteristics that we found in the analyzed projects. Software projects can contain further characteristics as well and the PaLiMod model can be extended to cover these additional cases. An example scenario is a commit that addresses multiple issues at once, but references only one of them. Thus, the existing linking is correct, but since the commit addresses multiple issues, the linking does not tell the whole truth. Such a characteristic is a candidate for extension and a potential subject for future work.

IV. Discovering Loners and Phantoms

The investigation on the Apache survey data showed that linked commit and issue data follow certain characteristics. We examined the characteristics of Loners and Phantoms, and

[2]https://www.openhub.net/

[3]https://issues.apache.org/jira/browse/HIVE-6782

[4]https://issues.apache.org/jira/browse/AMBARI-3412

[5]http://git-wip-us.apache.org/repos/asf/ambari/commit/f2146a41

[6]https://issues.apache.org/jira/browse/CAMEL-7354

[7]http://git-wip-us.apache.org/repos/asf/camel/commit/14cd8d6b

TABLE IV
SHARE OF COMMITS (IN %) CATEGORIZED BY THE CHANGED FILES AND MAXIMUM VALUES ARE GRAY COLORED.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Mean	SD
Linked	**64.0**	**98.3**	**60.6**	**44.4**	**48.6**	**84.1**	**90.0**	**95.2**	**90.4**	**80.9**	**94.1**	**61.4**	**40.1**	**89.7**	**72.6**	**74.3**	**19.1**
Source	30.4	3.5	34.4	46.7	44.6	0.6	37.9	29.3	48.3	28.0	78.9	33.0	28.9	34.5	14.1	32.9	18.3
Test	15.5	0.7	6.0	4.2	3.8	0.3	15.7	9.3	11.0	2.4	6.2	4.1	2.2	3.0	6.2	6.0	4.7
Admin	16.7	81.9	10.2	4.6	12.8	27.2	14.0	10.5	9.4	45.3	3.4	38.3	22.8	30.0	26.6	23.6	19.5
Combination	37.4	13.0	49.4	44.5	38.9	72.0	32.4	50.9	31.2	24.3	11.4	24.6	46.0	32.5	53.1	37.4	15.5
Non-Linked	**36.0**	**1.7**	**39.4**	**55.6**	**51.4**	**15.9**	**10.0**	**4.8**	**9.6**	**19.1**	**5.9**	**38.6**	**59.9**	**10.3**	**27.4**	**25.7**	**19.1**
Source	14.4	2.3	17.8	35.3	7.1	6.0	13.6	1.9	41.4	21.9	39.7	12.0	39.8	45.3	18.7	21.1	14.7
Test	29.4	3.5	23.4	11.4	12.5	1.8	11.7	0.0	11.9	11.7	1.3	4.1	5.6	11.3	14.3	10.3	7.9
Admin	43.8	57.0	36.6	31.9	69.9	54.1	59.9	11.2	37.0	53.2	53.8	78.3	30.7	37.7	50.5	47.0	16.3
Combination	12.3	37.2	22.2	21.4	10.5	38.1	14.8	86.9	9.7	13.2	5.1	5.6	23.9	5.7	16.5	21.5	20.1
Ratio	**0.6**	**0.0**	**0.7**	**1.3**	**1.1**	**0.2**	**0.1**	**0.1**	**0.1**	**0.2**	**0.1**	**0.6**	**1.5**	**0.1**	**0.4**	**0.3**	**1.0**
Source	0.5	0.7	0.5	0.8	0.2	10.0	0.4	0.1	0.9	0.8	0.5	0.4	1.4	1.3	1.3	0.6	0.8
Test	1.9	5.0	3.9	2.7	3.3	6.0	0.7	0.0	1.1	4.9	0.2	1.0	2.5	3.8	2.3	1.7	1.7
Admin	2.6	0.7	3.6	6.9	5.5	2.0	4.3	1.1	3.9	1.2	15.8	2.0	1.3	1.3	1.9	2.0	0.8
Combination	0.3	2.9	0.4	0.5	0.3	0.5	0.5	1.7	0.3	0.5	0.4	0.2	0.5	0.2	0.3	0.6	1.3

TABLE V
SHARE OF RESOLVED ISSUES (IN %) CATEGORIZED BY THE CHANGED TYPE AND MAXIMUM VALUES ARE GRAY COLORED.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	Mean	SD
Linked	**39.4**	**87.2**	**62.3**	**61.3**	**36.2**	**24.4**	**33.5**	**43.9**	**64.5**	**52.5**	**36.6**	**62.5**	**45.2**	**54.1**	**39.8**	**49.6**	**15.5**
Bug	65.9	57.9	41.0	65.8	61.2	51.2	53.0	60.8	48.2	41.1	70.5	51.3	53.1	37.0	43.5	53.4	9.8
Feature	33.0	41.9	56.4	34.1	37.7	46.3	40.8	38.5	51.5	35.7	29.3	47.9	45.1	59.4	56.5	43.6	9.0
Other	1.1	0.2	2.6	0.2	1.0	2.5	6.3	0.6	0.3	23.2	0.2	0.9	1.9	3.6	0.0	3.0	5.6
Non-Linked	**60.6**	**12.8**	**37.7**	**38.7**	**63.8**	**75.6**	**66.5**	**56.1**	**35.5**	**47.5**	**63.4**	**37.5**	**54.8**	**45.9**	**60.2**	**50.4**	**15.5**
Bug	69.2	51.3	33.6	62.9	59.6	49.7	43.4	54.6	35.5	55.7	75.2	43.2	43.9	23.6	61.7	50.9	13.5
Feature	29.7	47.4	65.6	36.0	40.2	47.8	52.1	43.6	63.2	36.5	24.2	55.5	53.1	67.9	38.3	46.7	12.6
Other	1.1	1.3	0.8	1.1	0.2	2.5	4.6	1.7	1.3	7.8	0.6	1.2	3.0	8.6	0.0	2.4	2.5
Ratio	**1.5**	**0.1**	**0.6**	**0.6**	**1.8**	**3.1**	**2.0**	**1.3**	**0.6**	**0.9**	**1.7**	**0.6**	**1.2**	**0.8**	**1.5**	**1.0**	**1.0**
Bug	1.1	0.9	0.8	1.0	1.0	1.0	0.8	0.9	0.7	1.4	1.1	0.8	0.8	0.6	1.4	1.0	1.4
Feature	0.9	1.1	1.2	1.1	1.1	1.0	1.3	1.1	1.2	1.0	0.8	1.2	1.2	1.1	0.7	1.1	1.4
Other	1.0	6.5	0.3	5.5	0.2	1.0	0.7	2.8	4.3	0.3	3.0	1.3	1.6	2.4	N/A	0.8	0.5

propose heuristic approaches to automatically interlink commit and issue data with these characteristics. Loners and Phantoms can be quantified by a retrospective analysis of the data in the VCS and the issue tracking platform.

A. The Loner Heuristic

The aim of this heuristic is to discover issues which get fixed by a single commit. It tries to detect candidates based on a combination of time and committer information and discards wrong candidates using commit and reopen history. The heuristic is specified as follows:

$$\forall x, \forall y, \exists z \Big(\big(ResolvedCR(x) \wedge NonLinkedCommit(y) \wedge CRResolvedEvent(z) \wedge HasCRElement(x,z) \wedge NonLinked \wedge PersonCond \wedge TimeCond \wedge ReopenCond \wedge CommitCond \big) \implies LinkCandidate(x,y) \Big)$$

$$NonLinked := \neg\exists w \big(Commit(w) \wedge isExplicitlyLinked(x,w) \big)$$

The results of Loner heuristic are pairs (x, y) of resolved change requests x and not linked change sets y, if all conditions, i.e. *PersonCond*, *TimeCond*, *ReopenCond*, and *CommitCond*, are satisfied.

$$PersonCond := hasResolved(x) = isCommittedBy(y)$$

The *person* condition (*PersonCond*) requires that the issue resolver, i.e. the person who resolved the change request x, is the same person as the change set committer, i.e. the person who committed the change set y.

$$TimeCond := resolvedTS(x) - commitTS(y) < ParTime \wedge resolvedTS(x) - commitTS(y) > 0$$

The *time* condition (*TimeCond*) specifies the maximum time-span between a commit and an issue resolution and is based on the parameter *ParTime*. We use a time frame of five minutes for the parameter as the analysis of the 15 projects has shown that 55.4% of the issues in the 1:1 case are resolved within 5 minutes after the commit.

$$CommitCond := \neg\exists v \big(Commit(v) \wedge isCommittedBy(v) = isCommitted(y) \wedge v \neq y \wedge commitTS(v) > commitTS(y) \wedge commitTS(v) < resolvedTS(x) \big)$$

The *commit* condition (*CommitCond*) checks that there are no commits addressing other issues between a commit and an issue linking candidate. Thus, there shall be no other commit

v by the committer of y which lies between the commit time of y and the resolve time of x. The condition is required, because otherwise no exact matching of commits to issues would be possible. Imagine a second not linked commit within the considered time frame, the issue could then also have multiple associated commits (*Phantoms*).

$$ReopenCond := \neg\exists u\big(CRResolvedEvent(u) \\ \wedge hasCRElement(x,u) \wedge u \neq z\big)$$

The *reopen* condition (*ReopenCond*) ensures that reopened issues are not taken into account. It does this by checking the non-existence of an arbitrary resolved event u of the change request x, which has to be different than z. If an issue has multiple resolve-events, i.e. it has been reopened at least once, it contains very probably further patches and thus, the issue would belong to the 1:n scenario again and has to cope with *Phantom* commits.

The remaining term in the heuristic's specification, *Non-Linked*, guarantees that the resolved change request has no associated commits and is therefore not linked. It achieves this by checking for non-existence of a commit linked to the change request. The first four predicates of the heuristic ensure that a right set of candidates is chosen, i.e. a resolved change request *x*, which has an associated change request resolved event *z*, and a non-linked commit *y*. If this set of candidates satisfies all conditions a *LinkCandidate* is found.

The used functions and predicates are derived by the annotations of the model. For example, the function *isCommittedBy(y)* can be seen as a getter returning the person which has contributed the commit y. Similarly, the predicate *HasCRElement(x,z)* checks, whether there is the relation *hasCRElement* between the change request x and the change request resolved event z. Predicates such as *NonLinkedCommit(y)* simply check for a specific class membership, in the given example if y is a non-linked commit. The function *commitTS* returns the point in time of the commit. Similarly, *resolvedTS* returns the timestamp of the issue-resolution. The comparison of identities is achieved by comparing names and email addresses.

B. The Phantom Heuristic

The aim of this heuristic is to discover non-linked commits (Phantoms) that address issues with at least one associated linked commit. An existing link between a commit and a change request serves as the baseline of the approach. It detects potential commits that address the same change request based on the commit time and the touched resources. The heuristic is specified as follows:

$$\forall x, \forall y\Big(\big(NonLinkedCommit(x) \wedge LinkedCommit(y) \wedge \\ PersonCond \wedge TimeCond \wedge ResCond\big) \\ \implies LinkCandidate(x,y)\Big)$$

The results of the Phantom heuristic are pairs (x, y) of linked commits x and non-linked commits y, if all conditions, i.e. *person*, *time*, and *resource*, are satisfied.

$$PersonCond := isCommittedBy(x) = isCommittedBy(y)$$

The *person* condition (*PersonCond*) requires that the committer of a non-linked candidate x has to be the same person as the committer of the baseline commit y.

$$TimeCond := abs(commitTS(x) - commitTS(y)) < ParTime$$

The *time* condition (*TimeCond*) ensures that the maximum time-span between a non-linked candidate commit and the baseline commit is *ParTime* days, being *ParTime* a specifiable parameter. Typically, more complex issues are not solved using single commits, thus multiple code fragments are committed, usually scattered across one or more work days. In order to take such behavior into account, the time frame for the Phantom heuristic, i.e. the value of *ParTime*, is set to 5 days, which is small enough to exclude unrelated commits and big enough to include potentially interesting commits.

$$ResCond := similarity(hasResource(x), hasResource(y)) \\ > ResOverlap$$

Finally, the *resource* condition (*ResCond*) requires that the lists of changed resources of two commits have to overlap of at least *ResOverlap* percent. The analysis of existing links in this 1:n scenario has shown that there is an overlapping of 65% of the commits' associated resource files. For this reason we have set the parameter *ResOverlap* for resource coverage to 66%.

The predicates *NonLinkedCommit(x)* and *LinkedCommit(y)* ensure that the heuristic is applied on pairs of non-linked commits and linked commits. If all conditions match, a *LinkCandidate* is found. Similar to the Loner heuristic, functions and predicates are highly related to the model's subjects and annotations. Additionally, the function *abs* is used, which returns the absolute value of the given number. It is required to ensure that a commit lies within the considered time frame. The function *similarity* takes two lists of file names and calculates the resource coverage by dividing the number of occurrences of the larger list's elements in the smaller one by the total number of elements in the larger list. File names are provided by the function *hasResource*, which takes a change set as argument and returns a list of its associated file names.

V. Evaluation

A central aim of our approach is to reduce the residual of non-linked commit and issue data in software projects, which have interlinking guidelines established. For the evaluation, we used a scenario-based approach to simulate different residuals of non-linked commit and issue data. Such a simulation was necessary because to the best of our knowledge no baseline dataset is publicly available for partial interlinked software projects. For each scenario, we randomly removed a percentage (10%, 20%, 30%, and 40%) of existing links from a project's dataset. The resulting set of commit and issue data built the input for the evaluation runs and the original dataset built the baseline used for the precision and recall calculation.

The random removal of links was done automatically and took care of certain constraints to avoid impossible interlinking constellations in the resulting datasets. For example, a random removal without constraints in case of a Phantom might unlink all commits of an issue, which results in a constellation that no longer fits the definition of a Phantom. For the random link removal we applied different combinations of three constraints that are based on the findings of our project survey (see Section III-B). The first constraint is the *link* constraint to ensure that at least one link to an issue with multiple links remains in case of a Phantom scenario. The *time* constraint is a constraint that shall avoid cases in which multiple, but time independent links to one issue get removed. For example, an issue gets re-opened after half a year. Such a situation leads to additional commits that address the same issue but are time independent. The third constraint, i.e. *person* constraint, ensures that the issue's resolver also committed the patch (Loner heuristic), or that the committer of the linked commit is the same as the committer of the not linked commit (Phantom heuristic). This constraint is especially important for scenarios with Loners, because of the limited amount of attributes that are available to detect and confine them from other scenarios, such as the Phantom.

We ran the evaluation for each heuristic and each of the different scenarios ten times to mitigate the risk of potential outliers introduced by the random manner of the deletion process and the used constraints. A Java application was used for the automatic execution of the evaluation runs and the collection of the respective results.

Overall, the evaluation results showed that both heuristics achieve high precision and recall. Unless otherwise stated, the values presented here were extracted from the evaluation runs based on 30% removed existing links. This rate is based on the average percentage of non-linked commits which is about 26% as presented in Section III-B. In case of the Loner heuristic, the overall precision is 96% with a recall of 92%. In case of the less restrictive *time* constraint, the precision remains stable and the recall decreases to 70%. The F-measure reduces from 0.94 to 0.81. For the Phantom heuristic, the overall precision is 73% with a recall of 53%. Moving from the combined *time-link* constraint to the single *link* constraint leads to a precision of 68% and a recall of 39%. The F-measure decreases from 0.61 to 0.50.

A. Loner Heuristic

Figure 3 illustrates the average results of the evaluation of the Loner heuristic after 10 runs under the *time* constraint. The heuristic achieves in 5 projects precision and recall values beyond 80%. At a first glance, CXF (P4) seems to be an outlier where the Loner heuristic does not work at all. The bad performance of the Loner heuristic in case of CXF is caused by a too small set of one-to-one mappings between commits and issues. There exist 25 links suited as candidates for deletion (unlinking), but when applying the *time* constraint this number shrinks to only three remaining candidates. Therefore our application for pursuing the evaluation did not remove any

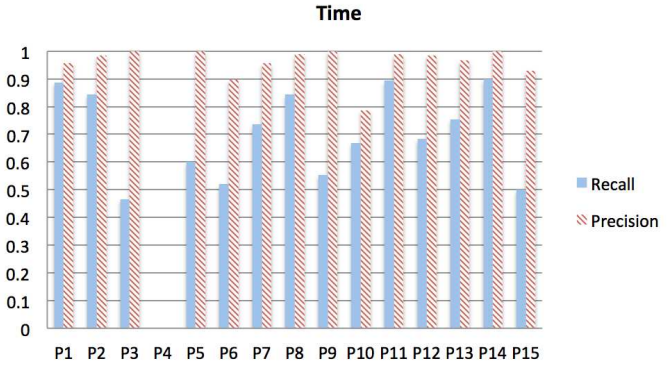

Fig. 3. Loner: *time* constraint

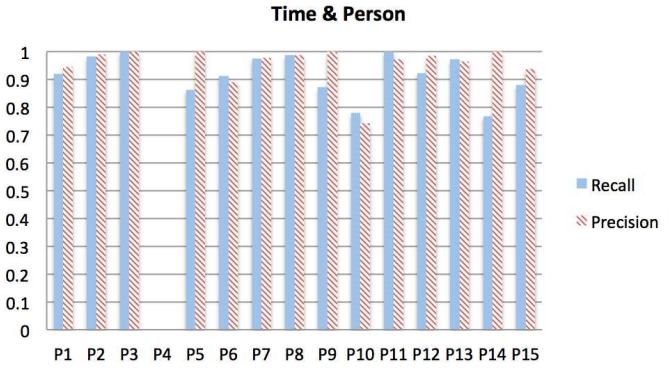

Fig. 4. Loner: combined *time-person* constraint

links as even a single link representing 33% of the delete candidates exceeds the specified threshold of 30%. In the 40% removal setting, in each run one link is deleted and in 80% of the cases correctly reestablished by our heuristic. Another interesting project is Camel (P3), which has a high precision and a rather low recall value under the *time* constraint. After a manual analysis of the data we found that one project member is responsible for approximately 45% of the commits. Due to the usage of different email addresses by this developer on the two platforms that were used to fill our model with data, i.e. version control and issue tracker, our heuristic's person condition (based on name and email information) cannot establish a matching between these identities and therefore almost every second deleted link cannot be re-discovered for this project. As previous work has shown, the problem of identity matching is more than an isolated case and happens quite frequently in Apache Software Foundation projects and others [12]. The Loner heuristic does not make use of existing links, therefore the results vary only slightly in the different test runs (10%, 20%, 30%, and 40% deleted links), involving the precision values being nearly constant and a variation of about 2% regarding recall. Given the application of the combined *time-person* constraint in the Loner heuristic (see Figure 4), increased recall values can be achieved which is based on the fact that this variant strongly covers the heuristic's conditions.

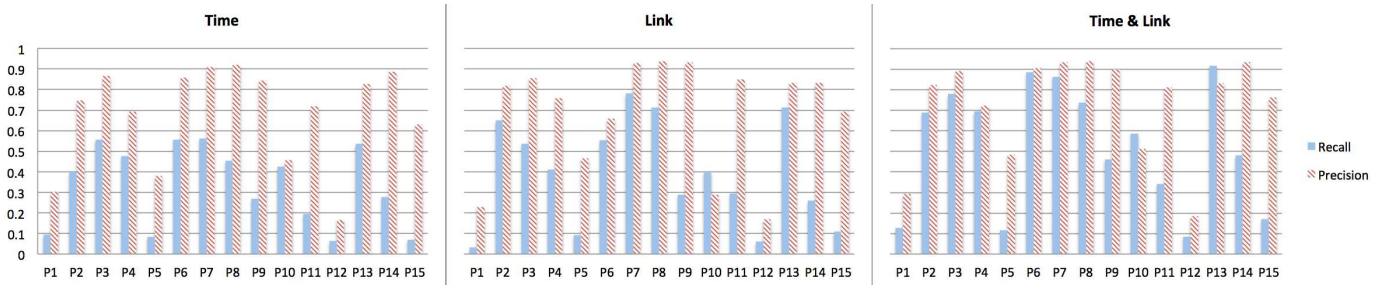

Fig. 5. Phantom: time, link, and time-link constraints

B. Phantom Heuristic

As depicted in Figure 5, the results of the Phantom heuristic are subject to higher fluctuations. For some projects high precision and recall values are achieved, while for a few projects both precision and recall values are rather low. Even the combination of time and link constraints does not lead to a significant improvement for certain projects. In general, we observe that the combined variant leads to smaller gaps between precision and recall values as can be clearly seen between the time, link, and time/link charts for the Phantom heuristic in Figure 5.

The Phantom heuristic uses existing links to identify candidates. Thus, the more links are removed during our deletion process, the smaller the base of existing links to learn from is, for example for determining the source files which are frequently updated. This fact is reflected in the decrease of the recall value which drops from 47% in the 10% setting to 27% when removing 40% of the links. The *link* constraint in our simulation mitigates this behavior and reduces this effect from 20% down to 4%.

We conducted further experiments with the Phantom heuristic to better understand and improve the results. Therefore, we tested different parameter settings for the heuristic's conditions, strengthened, weakened or removed some of them in order to see how the results evolve. First, we adjusted the person condition such that we require the committer of the non-linked candidate to be part of the issue's discussion by contributing at least one comment. Under the *link* constraint the modified heuristic detected less candidates, the precision improved by 4%, but the recall dropped by 10% on average. Second, we removed the person condition completely. Again, we used the *link* constraint in the 30% setting to compare the results. The number of identified candidates increased by 21%, the precision decreased by 5% and the recall increased about 3%. In a next step we combined this setting with a time-span of the heuristic's time condition broadened from 5 to 7 days, thus setting the parameter *ParTime* to 7. Recall values remained unchanged, but precision values decreased by further 2%. In total we suffered the loss of 7% precision and gained just 3% in recall. The F-measure remained stable at 0.50.

Finally, we tackled the remaining resource condition as well. We set the rate to which degree changed resources of two commits have to overlap, i.e. the *ResOverlap* parameter, from 66% to 30%. Under the *link* constraint we increased recall

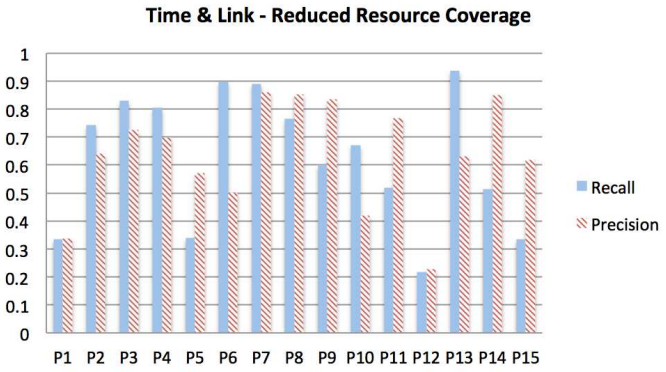

Fig. 6. Phantom: time-link constraint, resource coverage set to 30%

values by 8% on average and determined a decrease of 10% in precision. With a combined *time-link* constraint we observed an increase of 10% regarding recall and a drop of 9% in precision, so we ended up with 64% precision and 63% recall. The F-measure improved by 0.02 to 0.63. The results for the single projects are depicted in Figure 6.

VI. Discussion

Overall, the evaluation results showed that our proposed model and the proposed heuristics can be used to re-establish missing links in partly interlinked commit and issue data. We discuss the implications of the survey findings, the proposed heuristics and their performance, and the potential of the interlinking model.

A. The Partial Linking Model

The results of our evaluation showed that the PaLiMod model enables a fast integration of data from different data sources (e.g., VCS, issue tracking platform). The graph-based structure of the model provides a powerful mechanism to describe relationships and attributes of data entries independent of the data source. Currently, PaLiMod interlinks data from VCSs and issue tracking platforms, but it is possible to include discussions from mailing lists, chat protocols or other project related information. A further benefit of the PaLiMod model is data abstraction. For example, the definition of analysis algorithms or heuristics on top of the PaLiMod model can take place without knowledge about the actual structure of the data in the origin data source. In this paper, we evaluated the performance of the proposed heuristics with projects that

use Git and JIRA, but the heuristics would also work for data integrated from SVN or Bugzilla.

However, one limitation of the PaLiMod model, similar to other model-based approaches, is the transfer of new data (e.g., newly established links) back to the origin data source (e.g. VCS). We showed in our evaluation that the proposed heuristics and the PaLiMod model can be used to re-establish missing links, which can be exported into a data file or a report. As far as we know, there is no standardized way to integrate such heuristically established links into an existing issue tracking platform or into the history of a VCS.

B. Commit and Issue Interlinking

From the perspective of mining software repositories, developers process a stack of issues in a sequential order and every issue gets fixed by an explicit commit. However, the survey showed that such an ideal case is hard to find in practice. Patches and new features are contributed in arbitrary order and single-patch-fixes and other contributions tend to overlap.

Nevertheless, we were able to extract a set of interlinking characteristics used in the analyzed projects. We called them *Loner* and *Phantom*. A special challenge is the distinction of such characteristics in order to be strict enough to isolate from each other without refusing eligible candidates at the same time. Between the *Loner* and the *Phantom* the distinction of the characteristics is challenging especially in cases where two or more issues get fixed in concurrent. For example, commit C1 addresses issue I1 and commit C3 addresses issue I1. In between these two commits, the same developer commits another source change C2 without an issue key and changes the status of issue I2 to resolved. In such a case, the overlapping of the changed resources of the single commits can be used to determine the belonging of C2. Despite the information about the overlapping it is not possible to deterministically compute the belonging of such a commit.

C. Interlinking Heuristics

Both of the proposed interlinking heuristics perform well in most of the cases, however there are projects in which they don't because of various, often project specific, factors. The primary attribute used for all of the heuristics is the temporal relation of two events. For example, if Loners are within the specified time window there is a high chance that the heuristic will detect them and create a correct linking. Another important attribute of our approach is the matching of developer identities between identities used in the VCS and identities used in the issue tracking platform.

Moreover, the heuristics are based on the assumption that the source code contribution originates from the same person which changes the issue status to resolved. However, this assumption does not always hold and depends on the specific project. Especially the evolution of the recall values between Figures 3 and 4 reflects the influence of this assumptions onto the overall results. For example, for the project ActiveMQ (P1) the difference is small and our condition applies very well. In contrast, Hadoop (P6) has a divergence in its recall values which is, from our perspective, caused by code contributions and issue resolutions that are carried out by different persons.

During the experiments with various attributes and their settings, we identified the resource overlapping attribute and its threshold (*ResOverlap*) as the major impact factor on the Phantom heuristic. The threshold was determined empirically across all of the analyzed projects, which leads to better results for projects that have a low standard deviation. Our experiments have shown that the higher the overlapping of changed resources between the commits in a project are, the higher the recall. The downside of reducing the threshold is that for the projects with high overlapping, the precision drops sharply. For example, this can be recognized in projects Hadoop (P6) and Spark (P13) in which the precision decreases up to 40%. Nonetheless, there are projects, especially ActiveMQ (P1) and Sling (P12) where the Phantom heuristic does not perform well. The reasons are manifold and the mentioned resource condition is just one factor. Other factors are the general project organization, e.g. the number of active contributors, or even on the source code level, when multiple issues are fixed within a few days addressing only a small amount of resource files. In such cases, our heuristic tends to produce a lot of false positives, for example in the Sling project (P12).

It is important to highlight, that in difference to other research in this area, the proposed heuristics are very simple as they don't use rather complex techniques such as text-similarity to discover links. Our heuristics are based on simple information including developers, commits and issues which represent the essential elements in VCSs and issue tracking platforms. For example, they don't rely on artifacts like comments which might exist for certain issues, but usually not for all of them. Therefore, our heuristics are applicable to a broader set of cases, and despite of their simplicity, good results have been achieved.

VII. Threats to Validity

Empirical studies have limitations that have to be considered when interpreting their results. Our study is amenable to threats to the external and internal validity.

External Validity. For the extraction of the commit and issue interlinking, we relied on data gathered by mining source code repositories and issue tracking platforms of 15 different Apache projects. We limited the data extraction to a period of one year. These decision might limit the generalizability of our results, and further studies might need to be conducted to verify that our results can also be applied to other projects. However, to mitigate this risk, we have chosen the projects used in our analysis in a way to get a broad sample of various projects with different characteristics. Another threat that might limit the generalizability of our results is the use of only one type of VCS (Git) and only one type of issue tracking platform (JIRA). A different scope of functionality provided by a VCS or an issue tracking platform can lead to a different interlinking behavior.

Internal Validity. We analyzed interlinking characteristics of 15 Apache software projects and derived a set of interlinking

heuristics based on these findings. Based on this approach, we were able to extract the most prominent interlinking characteristics used in projects. There might be further interlinking characteristics, which only occur in single projects of our dataset. We tried to mitigate this by ensuring a large coverage of the links found in our dataset by the proposed heuristics.

The dataset used for the evaluation of the interlinking heuristics was derived from the original dataset used in the survey. This reuse of the dataset may influence the performance measurements of the heuristics during the evaluation. One possibility to overcome this threat can be the use of a dataset, in which the residual of non-linked commit and issue data was manually annotated by an active member of the according project. Due to the absence of such a manual annotated dataset, we minimized this threat by a random removal of links from interlinked commit and issue data. Furthermore, we repeated each evaluation ten times to mitigate a bias introduced by the random removal.

VIII. Related Work

The interlinking of development artifacts (e.g., commits, issues, emails) has been addressed multiple times in related literature. In the following, we discuss the most relevant related work in the area of commit and issue interlinking.

Mockus and Votta [7] introduced an early approach to extract a textual description of the purpose of a change set from a VCS only. One of the first approaches that uses data from a VCS in combination with data from an issue tracker was presented by Fischer et al. [13]. The incentive of their work was the research on software evolution, which relies on detailed and rich data of software projects.

Śliwerski et al. [14] introduced an approach to automatically detect changes that fix a certain bug. Their approach is built on top of an interlinked commit and issue dataset. Kim et al. [15] proposed an approach to automatically detect bug-introducing patterns based on data mined from fixed bugs. They interlinked commit and issue data for the change pattern extraction.

Ayari et al. [16] addressed threats on building models based on commit and issue data. They showed that a considerable number of links cannot be established based on a numerical issue id in the commit message. Bird et al. [17] investigated the impact of bias in commit and issue datasets used in research, such as in the bug prediction area. Their experiments showed that bias indeed influences the performance of bug prediction approaches, such as BugCache [18]. One proposed way to overcome this bias is the use of explicit linking mechanisms as offered by platforms, such as *Rational Team Concert*.[8] Bachmann and Bernstein [6] proposed a set of software engineering process measurements based on commit and issue data. A survey of five open source projects and one closed source project showed major differences in the engineering process expressed by the commit and issue data. In further work [2], Bachmann et al. established a ground truth of commit and issue data from the Apache Httpd project created by a core developer of the project. Their analysis showed that bugs get often fixed without an issue entry and that commits not always change functionality of a software system. They introduced a tool call Linkster [19], which provides capabilities for manual commit and issue interlinking. A series of approaches [9], [8], [20] tried to overcome the raised restrictions (e.g., [18], [6], [2]) of data interlinking based on numerical issue ids only and proposed interlinking heuristics based on, for example, text similarity (e.g., [9]) or change set overlapping (e.g., [8]).

There are other research areas that address the interlinking of source code artifacts other than commit and issues. For example, Bachelli et al. [21], [22] investigated the interlinking of email data from mailing lists with commit or patch data. The area of traceability research (e.g., [23], [24]) addresses the interlinking of source code changes with documentation changes.

Our approach differs from the mentioned related work, as we address the interlinking of commit and issue data in environments that use already interlinking, but have a residual of non-linked commit and issue data. The unused information stored in such a residual can foster approaches (e.g., bug triaging), which heavily rely on the mined data of VCS and issue tracking platforms. To best of our knowledge, such a scenario is not covered by any existing work.

IX. Conclusion

The interlinking of commit and issue data has become a de-facto standard in software projects, which is reflected in guidelines of large open-source communities [1]. In this work, we (1) investigated the extent to which developers follow such interlinking guidelines by a survey of 15 Apache projects, (2) deducted a set of interlinking characteristics found in the surveyed projects, and (3) proposed a model (ParLiMod) and a set of heuristics to interlink the residual of non-linked commits and issues.

We observed that in the majority of the analyzed projects, the number of commits linked to issues is higher than the number of commits without link. On average, 74% of commits are linked to issues and 50% of the issues have associated commits. Based on the survey data, we identified two interlinking characteristics which we call *Loners* (one commit, one issue) and *Phantoms* (multiple commits, one issue). For these two characteristics, we proposed heuristics to automatically interlink non-linked commit and issue data. The evaluation results showed that our approach can achieve an overall precision of 96% with a recall of 92% in case of the Loner heuristic and an overall precision of 73% with a recall of 53% in case of the Phantom heuristic.

Potential future work includes the analysis of further software projects to allow for a quantitative description of the additional characteristic (one commit addresses multiple issues) we have discovered, but have not tackled yet. The model provides also a solid basis for the development of more complex heuristics, including e.g., text-similarity.

[8]https://jazz.net/products/rational-team-concert/

REFERENCES

[1] Apache Software Foundation, "How should i apply patches from a contributor?" [Online]. Available: http://www.apache.org/dev/committers.html

[2] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 97–106. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882308

[3] B. A. Romo, A. Capiluppi, and T. Hall, "Filling the gaps of development logs and bug issue data," in *Proceedings of The International Symposium on Open Collaboration*, ser. OpenSym '14. New York, NY, USA: ACM, 2014, pp. 8:1–8:4. [Online]. Available: http://doi.acm.org/10.1145/2641580.2641592

[4] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 90–. [Online]. Available: http://dl.acm.org/citation.cfm?id=950792.951355

[5] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 298–308. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070530

[6] A. Bachmann and A. Bernstein, "Software process data quality and characteristics: A historical view on open and closed source projects," in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, ser. IWPSE-Evol '09. New York, NY, USA: ACM, 2009, pp. 119–128. [Online]. Available: http://doi.acm.org/10.1145/1595808.1595830

[7] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on*, 2000, pp. 120–130.

[8] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 63:1–63:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393671

[9] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: Recovering links between bugs and changes," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 15–25. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025120

[10] M. Brandtner, S. C. Müller, P. Leitner, and H. C. Gall, "Sqa-profiles: Rule-based activity profiles for continuous integration environments," in *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER '06, 2015.

[11] "Project web-site of the *PaLiMod* model." [Online]. Available: http://www.ifi.uzh.ch/seal/people/schermann/projects/commit-issue-linking.html

[12] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 137–143. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138016

[13] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, Sept 2003, pp. 23–32.

[14] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[15] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 81–90. [Online]. Available: http://dx.doi.org/10.1109/ASE.2006.23

[16] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta, "Threats on building models from cvs and bugzilla repositories: The mozilla case study," in *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '07. Riverton, NJ, USA: IBM Corp., 2007, pp. 215–228. [Online]. Available: http://dx.doi.org/10.1145/1321211.1321234

[17] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595716

[18] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 489–498. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.66

[19] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein, "Linkster: Enabling efficient manual inspection and annotation of mined data," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 369–370. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882352

[20] T. Bissyande, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Reveillere, "Empirical evaluation of bug linking," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, March 2013, pp. 89–98.

[21] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes, "Benchmarking lightweight techniques to link e-mails and source code," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 205–214. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2009.44

[22] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 375–384. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806855

[23] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Information retrieval models for recovering traceability links between code and documentation," in *Software Maintenance, 2000. Proceedings. International Conference on*, 2000, pp. 40–49.

[24] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, "Recovering traceability links between source code and fixed bugs via patch analysis," in *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, ser. TEFSE '11. New York, NY, USA: ACM, 2011, pp. 31–37. [Online]. Available: http://doi.acm.org/10.1145/1987856.1987863